INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A floating-point library for integer processors

C. Bertin — N. Brisebarre — B. Dupont de Dinechin — C.-P. Jeannerod — C. Monat — J.-M. Muller — S. Raina — A. Tisserand

N° 5268

July 2004

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5268--FR+ENG

A floating-point library for integer processors

C. Bertin* , N. Brisebarre† , B. Dupont de Dinechin* , C.-P. Jeannerod† ,
C. Monat* , J.-M. Muller† , S. Raina† , A. Tisserand†

Thème SYM — Systèmes symboliques
Projet Arénaire

Rapport de recherche n° 5268 — July 2004 — 17 pages

Abstract: This paper presents a C library for the software support of single precision floating-point (FP) arithmetic on processors without FP hardware units such as VLIW or DSP processor cores for embedded applications. This library provides several levels of compliance to the IEEE 754 FP standard. The complete specifications of the standard can be used or just some relaxed characteristics such as restricted rounding modes or computations without denormal numbers. This library is evaluated on the ST200 VLIW processors from STMicroelectronics.

Key-words: Computer arithmetic, floating-point arithmetic, addition, multiplication, division, square-root, integer processor, VLIW, DSP

* STMicroelectronics. 12 rue Jules Horowitz, F–38019 GRENOBLE, FRANCE
† Arénaire project (CNRS – ENS Lyon – INRIA – UCBL). LIP, ENS Lyon. 46 allée d'Italie. F–69364 LYON Cedex 07, FRANCE

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Une bibliothèque flottante pour processeurs entiers

Résumé : Ce papier présente une bibliothèque C pour le support de l'arithmétique flottante simple précision sur des processeurs sans unité flottante matérielle comme des coeurs de processeurs VLIW ou DSP embarqués. Cette bibliothèque propose différents niveaux de compatibilité à la norme IEEE 754. Les spécifications complètes de la norme peuvent être utilisées, ou bien seulement une partie comme des modes d'arrondi restreints ou des calculs sans les nombres dénormalisés. Cette bibliothèque a été évaluée sur les processeurs VLIW ST200 de STMicroelectronics.

Mots-clés : Arithmétique des ordinateurs, arithmétique flottante, addition, multiplication, division, racine carrée, processeur entier, VLIW, DSP

1 INTRODUCTION

Most embedded systems for digital signal processing, image processing and digital control rely on *integer* or *fixed-point processors* [1]. Indeed, for those applications, most fast and low-power processor cores have no FP unit for cost reasons (smaller area). When implementing algorithms dealing with real numbers on such processors, one has to introduce some scaling operations, in the target program, in order to keep accurate computations [2]. The insertion of scaling operations is complicated due to the wide range of real numbers required in many applications and it depends on the algorithm and data. Furthermore, scaling is time consuming at both the application and design levels. Algorithms based on FP arithmetic do not have this problem. Then, porting programs from general purpose processors (with FP unit) to integer or fixed-point cores is a complex task. For instance, prototyped applications using Matlab have to be converted to fixed-point format. In those cases, an efficient solution may be to use a fast FP emulation layer over the integer or fixed-point units. This is the goal of our work.

The C library, presented in this paper, provides the basic five operations: addition, subtraction, multiplication, division and square-root for a quasi-fully compliant single-precision (SP) IEEE 754 FP format (the flags are not supported). But this library also provides some running modes with relaxed characteristics: no denormal numbers or restricted rounding modes for instance. A secondary goal of this work is to study the impact of the more or less full compliance to IEEE floating-point standard on software implementations. This library has been developed and validated within a collaboration with STMicroelectronics. The library has been targeted to the VLIW (very long instruction word) processor cores of the ST200 family.

This paper is organized as follows. A summary on floating-point arithmetic is presented in Section 2. In Section 3, the target processor and compiler are presented. The new library is presented in Section 4. The validation of this new library is discussed in Section 5. The results of the implementation of some typical applications are presented in Section 6 as well as a comparison to the original library.

2 FLOATING-POINT ARITHMETIC

The FP standard defines the data format, some special values, the possible rounding modes, conversion rules, the behavior and the accuracy of the basic five operations ($\pm, \times, /, \sqrt{}$). In this section, we recall the main features of the single precision IEEE FP 754 standard [3] and the corresponding basic operations (see [4] for instance).

2.1 Format of Single Precision Floating-Point Numbers

A real number x is represented, in the floating-point format (the `float` data type in the C programming language for single precision numbers), using 3 fields: the *sign* s_x, the *exponent*

e_x and the *mantissa* m_x such that:

$$x = (-1)^{s_x} \times m_x \times 2^{e_x}$$

The sign s_x is a 1-bit field: $s_x = 0$ for positive values and $s_x = 1$ for negative values. The exponent e_x is represented using an 8-bit integer with $-126 \leq e_x \leq 127$. The stored exponent is represented using a biased representation with bias $b = 127$, i.e. the stored exponent is an unsigned integer $e_{x,b}$ such that $e_{x,b} = e_x + b$. The minimum and maximum biased exponents, 0 and 255, are reserved for zero, the special values and the denormal numbers. The mantissa, also known as *significand*, is composed of an *implicit* leading bit and the *fraction* f_x such that $m_x = 1.f_x$ for *normal* numbers and $m_x = 0.f_x$ for *denormal* numbers. In single precision, the fraction is a 23-bit field. The format is illustrated on Figure 1.

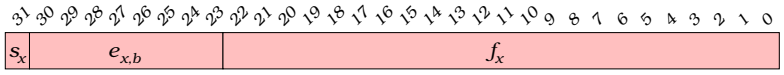

Figure 1: Internal format of IEEE 754 single precision floating-point numbers.

2.2 Representable Numbers and Special Values

Normal numbers are represented by $x = (-1)^{s_x} \times 1.f_x \times 2^{e_x}$ with $-126 \leq e_x \leq 127$. As the implicit leading bit is always 1, it is not stored and it is also called the *hidden* bit. For normal numbers, we have $m_x \in [1, 2)$.

The denormal, or *subnormal*, numbers are such that $e_{x,b} = 0$ but the fraction is non-zero (otherwise it would be the zero value). In this case, the implicit leading bit is 0 and the exponent e_x is set to $e_{min} = -126$. Denormal numbers are represented by $x = (-1)^{s_x} \times 0.f_x \times 2^{-126}$.

Due to the implicit leading one in the mantissa, the zero value is not representable. Zero is thus the special value such that with $e_{x,b} = f_x = 0$. The unspecified sign bit allows two values for zero: -0 and $+0$. This is consistent with the following properties : $\frac{1}{+0} = +\infty$ and $\frac{1}{-0} = -\infty$. The test $-0 = +0$ always returns a true value.

Infinity values are denoted by $e_{x,b} = 255$ and $f_x = 0$. The sign bit distinguishes between $-\infty$ and $+\infty$. Those values are used for overflows. The value NaN(*Not a Number*) is denoted by $e_{x,b} = 255$ and $f_x \neq 0$. It is used to represent a value that does not represent a real number, such as $0/0$, $\sqrt{-1}$ or $0 \times \infty$. In this work, we only consider one category of NaN(while the standard provides two different categories).

	s_x	e_x	f_x
$\sqrt{2}$	0	01111111	01101010000010011110011
2	0	10000000	00000000000000000000000
-3.25	1	10000000	10100000000000000000000
π	0	10000000	10010010000111111011011
1 000	0	10001000	11110100000000000000000
0.001	0	01110101	00000110001001001101111

	s_x	e_x	f_x
-0	1	00000000	00000000000000000000000
$+0$	0	00000000	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
NaN	0	11111111	00000000000000000000001
NaN	0	11111111	11111111111111111111111

Table 1: Examples of representable numbers and special values in the single precision floating-point format.

2.3 Rounding

If x and y are representable numbers (*machine numbers*), then the result of the operation $r = x \circledcirc y$ may not be representable in the target format (e.g. 1/3 in the decimal system). The result must be *rounded*. The standard provides four rounding modes:

- round to $+\infty$, denoted by $\triangle(r)$: returns the smallest machine number larger than or equal to the exact result r.
- round to $-\infty$, denoted by $\nabla(r)$: returns the largest machine number smaller than or equal to the exact result r.
- round to 0, denoted by $\mathcal{Z}(r)$: returns $\triangle(r)$ for negative numbers and $\nabla(r)$ for positive numbers.
- round to *nearest*, denoted by $\circ(r)$: returns the machine number that is the closest to the exact result r. If r is the exact middle between two consecutive machine numbers, it returns the FP number whose least significant bit is a 0 (*round to nearest even*). This is the default rounding mode in practice.

The standard requires an important property: *correct rounding*. Let x and y be two machine numbers, $\circledcirc$ an operation of $\{+, -, \times, \div\}$ and $\diamond$ the active rounding mode among the 4 IEEE rounding modes. The result of $(x \circledcirc y)$ must be $\diamond(x \circledcirc_{th} y)$: the returned result must be the result of the operation computed using an infinite accuracy (i.e. the theoretical operation $\circledcirc_{th}$) and then rounded. The standard requires the same property for the square root operation.

2.4 Conversion, Comparison and Behavior in IEEE Floating-Point Arithmetic

The standard defines conversion operations from integers towards floating-point numbers (FPN), from FPNs towards integers, from FPNs towards integers represented using a FPN, between FPNs of different formats and finally, between binary and decimal representations for input/output. The comparisons defined in the standard are: $=$, $>$, $<$ and non-ordered.

The standard specifies that "*no floating-point operation should alter the behavior of the overall computation*". Consequently, some flags are defined in order to inform the system about the behavior of some operations. Five flags are defined in the standard: *invalid* operation (the result is NaN), *overflow* (the result is $\pm\infty$ or the largest representable number depending on the rounding mode), *underflow* (the result is ± 0 or a denormal number), *division by zero* (the result is $\pm\infty$) and *inexact result* (raised when an operation involves a non-exact result).

3 STMICROELECTRONICS VLIW PROCESSOR AND COMPILER

The ST200 family of high performance low power VLIW processor cores are targeted at STMicroelectronics system on chips (SOC) solutions for use in computationally intensive applications as a host or an audio or video processor. Such applications include embedded systems in consumer, digital TV and telecommunication markets.

3.1 ST200 Family of VLIW Processor Cores

The ST200 is a four issue VLIW processor and is significantly simpler and smaller than an equivalent four issue super-scalar processor. The compiler is key to generate, schedule and bundle operations, removing the need to add complex hardware to achieve the same results.

The ST200 design originates from the LX research project started as a joint development by Hewlett-Packard laboratories and STMicroelectronics [5] done by Josh Fisher and his team for printer imaging applications. STMicroelectronics purchased the rights to develop the LX architecture [6] and to create the ST200 processor family.

Figure 2 displays the basic organization of a ST200 processor. This processor executes up to 4 operations per cycle, with a maximum of one control operation (goto, jump, call, return), one memory operation (load, store, prefetch) and two multiply operations per cycle. All arithmetic instructions operate on integer values, with operands belonging either to the General Register file (64×32-bit), or the Branch Register file (8×1-bit). The multiply instructions are restricted to 16×32-bit on the earlier core variants, but have been recently extended to 32-bit by 32-bit integer and fractional multiplication on the ST231 core. There is no divide instruction but a division step instruction suitable for 32-bit division. In order to reduce the use of conditional branches, the ST200 processor also provides conditional selection instructions.

The ST200 features a clean, compiler-friendly 6-stage instruction pipeline depicted in Figure 3, where all operations are fully pipelined. Almost all operation latencies are 1 cycle, meaning that the result of an operation can be used in the next *bundle*. Some instructions have a 3-cycle latency (load, multiply, compare to branch) or in one case a 4-cycle latency (load link register LR to call). Starting from the ST230 member of the ST200 family, latencies are enforced with *interlocks*, that enable some important code size reduction by

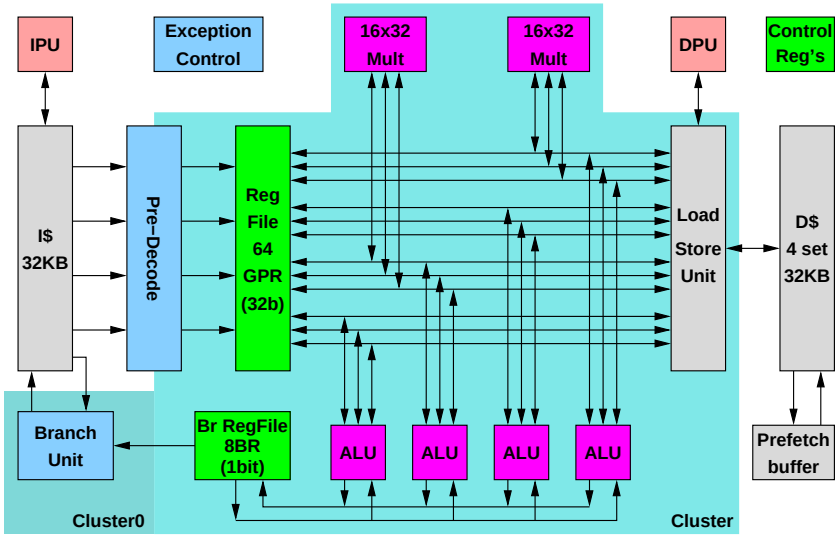

Figure 2: The ST200 VLIW processor architecture.

removing the *nops* or their equivalent that may be otherwise needed to guarantee a correct schedule.

3.2 The ST200 Compiler

STMicroelectronics developed the `st200cc` compiler based on the Open64 technology [7] and the LAO code optimizer [8], in order to replace the Multiflow Trace Scheduling compiler [9] originally used on the LX processor. The `st200cc` compiler was released in early 2003 and is currently actively developed to support new ST200 cores and advanced optimizations.

The STMicroelectronics ST200 production compiler `st200cc` is based on the GNU `gcc` and g++ compiler front-end components, the Open64 compiler technology [7], a global instruction scheduler adapted from the ST100 LAO code generator [8], an instruction cache optimizer [10] including support for dead code elimination (DCE), dead data elimination (DDE) and the GNU assembler, linker and binary utilities. The Open64 compiler has been re-targeted from the Intel IA64 to the STMicroelectronics ST200, except for the instruction predication, the global code motion (GCM), the instruction scheduler and the software pipeliner (SWP). Indeed these components appeared to be too dependent on the IA64 architecture predicated execution model and on its support of modulo scheduling through rotating register files.

In the `st200cc` compiler, the Open64 GCM and SWP components are replaced by the ST200 LAO instruction scheduler / software pipeliner, which is activated at optimization

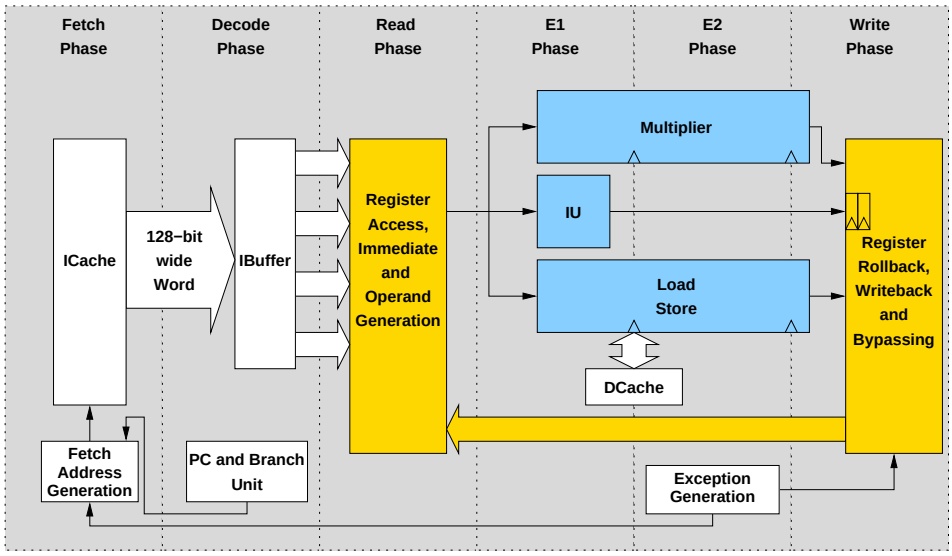

Figure 3: The ST200 processor pipeline.

level `-O3`. At optimization level `-O2` and below, only the Open64 basic block instruction scheduler is active.

In addition, access to integer and DSP arithmetic from C and C++ is supported by an extensive set of *intrinsic functions*. The functions defined in the ST200 Application Programming Interface provide access to low-level features, such as ST200 leading zero count instruction and to medium level features, such as the standard ETSI [11] and ITU [12] *basic operators*. All intrinsic functions are especially optimized by C and C++ compilers and are also delivered as standard C models for application validation on a workstation.

The intrinsic functions support has been fully developed in assembly language, validated using their C semantic models and a Data Generation Language [13] and regenerated through an automatic flow as expansion function in the Open64 CGIR intermediate representation. The compiler either generates open code or emits call to library functions, leading to very efficient 64 bits and DSP arithmetic. The low level integer arithmetic support directly uses the intrinsic functions.

3.3 Initial Floating-Point Library Derived from SoftFloat

The floating-point support is a derived work from John Hauser's SoftFloat package [14], using the 64-bit variant of the IEEE754 floating-point implementation, but keeping only one rounding mode (round to nearest even), removing denormal support and using specific intrinsic functions such as count leading zero to gain speed.

4 IMPLEMENTED LIBRARY

The presented new library implements single precision floating-point algorithms for the five basic operations $+$, $-$, $\times$, $/$ and $\sqrt{}$. Denormal numbers as well as the four rounding modes of Section 2.3 are available and the user may choose to use them or not. In practice, we have two versions of the library: one without the denormal numbers (labeled wo/D) and another one with the support of denormal numbers (labeled w/D). Both versions are useful depending on the target application. The input and output of all algorithms are single precision FP variables (representable numbers or special values). Currently the flags are not handled, like in the current FP support of the ST200. This will be dealt within a future work.

All basic operations can be decomposed into the four phases below:

Phase A: Decomposing operands. For an operand $x = x_{31} \cdots x_1 x_0$ formatted as in Section 2.1, its fraction f_x is defined as the 23-bit integer equal to $x_{22} \cdots x_1 x_0$; its biased exponent $e_{x,b}$ is defined as the 8-bit biased integer $e_{x,b} = e_x + b$ equal to $x_{30} \cdots x_{23}$ and with by $b = -127$; its sign s_x is defined as the bit x_{31}. During the computation, those three values are stored in 32-bit individual registers.

Phase B: Handling special values. If an operand is a special value such as $\pm\infty$, ± 0 or NaN, the result of the operation is known in advance and thus requires no numerical computation. Consider for example the case of multiplication, without denormals. (The same remark applies to other operations, with or without denormals.) The table below displays all the possible values of the product $\diamond(x \times y)$ when x or y is a special value.

x, y	NaN	$+\infty$	$-\infty$	$+0$	-0	other
NaN	NaN	NaN	NaN	NaN	NaN	NaN
$+\infty$	NaN	$+\infty$	$-\infty$	NaN	NaN	$\pm\infty$
$-\infty$	NaN	$-\infty$	$+\infty$	NaN	NaN	$\pm\infty$
$+0$	NaN	NaN	NaN	$+0$	-0	± 0
-0	NaN	NaN	NaN	-0	$+0$	± 0
other	NaN	$\pm\infty$	$\pm\infty$	± 0	± 0	$\diamond(x \times y)$

A first implementation of the above table can be a sequence of four "if-then-else" tests on the values of $e_{x,b}$ and $e_{y,b}$ (pseudo-code on the left). However, one can check that $\{e_{x,b}, e_{y,b}\} \cap \{0, 255\} \neq \emptyset$ with only one "if-then-else" test after a "max" computation (pseudo-code on the right). This follows from the fact that testing the equivalent condition $\{e_{x,b} - 1, e_{y,b} - 1\} \cap \{-1, 254\} = \emptyset$ is easy: indeed, in 2's complement representation $-a_{31}2^{31} + \sum_{i=0}^{30} a_i 2^i$, -1 and 254 are the only integers in the range $\{-1, 254\}$ such that $a_1 = a_2 = \cdots = a_7 = 1$. Now denote by $E_{x,i}$ and $E_{y,i}$ the bits of rank i for $e_{x,b} - 1$ and $e_{y,b} - 1$ respectively and let $E_x = \sum_{i=1}^{7} E_{x,i} 2^i$ and $E_y = \sum_{i=1}^{7} E_{y,i} 2^i$. We see that condition $\{e_{x,b}, e_{y,b}\} \cap \{0, 255\} \neq \emptyset$ is equivalent to condition $\max(E_x, E_y) = 254$. Hence the pseudo-code on the right. The goal here is to filter all the special values cases in only one test (and costly jump) since those values are extremely rare. On the target processor the min, max operations are executed in only one cycle.

if $e_{x,b} = 255$ **then**
  if $f_x \neq 0$ **then return** x;
  if $e_{y,b} = 255$ **and** $f_y \neq 0$ **then return** y;
  if $e_{y,b} = 0$ **then return** NaN;
  return $\pm\infty$;
if $e_{y,b} = 255$ **then**
  if $f_y \neq 0$ **then return** y;
  if $e_{x,b} = 0$ **then return** NaN;
  return $\pm\infty$;
if $e_{x,b} = 0$ **then return** ± 0;
if $e_{y,b} = 0$ **then return** ± 0;

$ME = \max(E_x, E_y)$;
if $ME = 254$ **then**
  if $e_{x,b} = 255$ **then**
    if $f_x \neq 0$ **then return** x;
    if $e_{y,b} = 255$ **and** $f_y \neq 0$ **then return** y;
    if $e_{y,b} = 0$ **then return** NaN;
    return $\pm\infty$;
  if $e_{x,b} = 0$ **then**
    if $e_{y,b} = 255$ **then return** NaN;
    return ± 0;
  return $\pm y$;

Phase C: Computing the result. Let $r = \diamond(x \circledcirc y) = (-1)^{s_r} \times 1.f_r \times 2^{e_r}$ and let $e_{r,b} = e_r + 127$. In general, the computation of s_r, $e_{r,b}$, f_r requires five substeps:

C1. Deduce s_r from s_x and s_y and "guess" $e_{r,b}$ from $e_{x,b}$ and $e_{y,b}$.
C2. Compute the mantissa $m_r = m_x \circledcirc m_y$.
C3. If $m_r \notin [1, 2)$, normalize m_r and update $e_{r,b}$.
C4. Round m_r according to the rounding mode $\diamond$.
C5. If $\diamond(m_r) \notin [1, 2)$, normalize m_r and update $e_{r,b}$.

In the case of denormal input numbers, the preceeding steps can be easily adapted. Those cases were filtered in phase B and dedicated code performs the computation in this case. Even if the input operands are normal numbers, a denormal result can occur during the last part of the computation, but its treatment is quite simple.

In the next four Sections (4.1 – 4.4), we will quickly describe the algorithms for each operation in this phase.

Phase D: Setting up the output. When the values of s_r, $e_{r,b}$, f_r are available, one can detect overflow/underflow by testing the value of $e_{r,b} = 0$ or $e_{r,b} = 255$. If $0 < e_{r,b} < 255$ the output is set up as the "well formated" 32-bit value $r = (s_r << 31) + (e_{r,b} << 23) + f_r$.

4.1 Floating-Point Addition and Subtraction

For operands x, y encoded by `X`, `Y` as in 2.1, $r = \diamond(x - y)$ is generated using an addition after complementing the first bit of `Y`. Hence we now focus on efficient implementation of FP addition $r = \diamond(x + y)$ only. First, $e_{r,b}$ is guessed to be `E` $= \max(e_{x,b}, e_{y,b})$ and the exponent difference $e_{x,b} - e_{y,b}$ is computed as a 32-bit signed integer `D`. Then, from the last 26 bits of `X` $<<$ 3 and `Y` $<<$ 3, mantissas `MX`, `MY` are built up as 32-bit signed integers in two's complement representation; if `D` < 0, these mantissas are further interchanged in order to reduce to the case where $e_x \geq e_y$.

Alignment of the two mantissas is done by right shifting `MY` by $m = \min(26, |\texttt{D}|)$ positions; to prepare for rounding, the last bit of this new aligned mantissa is further set to 1 if at least one of the last m bits of `MY` is nonzero (one part of the sticky bit computation). Mantissas

can now be added together: the first bit of the sum gives s_r and its absolute value `M` gives a first (possibly unormalized) guess for m_r. Mantissa normalization follows by left or right shifting `M`, depending on the number of the leading zeroes of `M` and by updating the last bit of the shifted `M` (as it was done for `MY`). Then $e_{r,b}$ is obtained by possibly incrementing `E` by one according to the active rounding mode $\diamond$, the values of the last three bits of `M` and the sign s_r. The rounding of `M` is done in a similar fashion [4].

Special values of operands are handled efficiently as described in phase B (Section 4) by testing if $\{e_{x,b}, e_{y,b}\} \cap \{0, 255\} \neq \emptyset$. The returned value then essentially depends on the sign of `D`; when both operands are zero, this value is either $+0$ or -0, depending on the rounding-mode (see standard [3]). Denormals are trapped as such special values ($e_{x,b}$ or $e_{y,b}$ is zero) and then handled specifically.

The efficiency of such an implementation of FP addition is shown in Table 2; the main reasons for it are:

1. the approach of phase B (Section 4) for detecting special values;
2. the use of the `MIN`, `MAX` functions rather than more costly classic "if-then-else" tests;
3. the use of a dedicated processor "leading zero count" instruction available in the ST200 family.

Table 2 reports the timing in number of bundles for the addition operation. This is the value of the average case timing (i.e. no special values and no over/under-flow). Two versions of the new library are compared to the original library. The first version is without denormal numbers (wo/D) and with rounding mode set to rounding to nearest even. The second version is with denormal numbers (w/D). Table 2 also reports the code size (in 32-bit words) and the bundle fill rate (i.e. the number of effectively executed instructions among all the possible 4 parallel instructions for each bundle).

	original lib.	new lib. wo/D	new lib. w/D
time (# bundle)	58 (100%)	37 (64%)	43 (74%)
code size	584 (100%)	872 (149%)	888 (152%)
bundle fill rate	47%	62%	67%

Table 2: Main characteristics of the addition operation.

4.2 Floating-Point Multiplication

The product of the two mantissas `MX` $\times$ `MY` is performed using the standard 4 partial product decomposition. Decompositions using only 3 partial product have been investigated without success. Indeed, those decompositions reduce the number of intermediate products but not the total latency and furthermore they require additional extraction instructions before the product.

Table 3 reports the timing, code size and bundle fill rate for the multiplication operation.

	original lib.	new lib. wo/D	new lib. w/D
time (# bundle)	44 (100%)	31 (70%)	35 (79%)
code size	592 (100%)	640 (108%)	840 (142%)
bundle fill rate	54%	71%	69%

Table 3: Main characteristics of the multiplication operation.

4.3 Floating-Point Division

We start by taking $s_r = \mathrm{XOR}(s_x, s_y)$ and $e_{r,b} = e_{x,b} + e_{y,b} - 127$ and by left shifting the mantissa of y by one position. In the normalized case, $2^{-1} < m_x/m_y < 2$ and we shall compute the first binary digits of $m_x/(2m_y) = 0.q_1q_2q_3\cdots$. More precisely, if $m_x < m_y$ then $q_1q_2 = 01$ and one needs the next 24 bits $q_3\cdots q_{25}q_{26}$, q_{26} being the guard bit used for rounding; if $m_x \geq m_y$ then $q_1 = 1$ and then only $q_2\cdots q_{24}q_{25}$ are needed, with q_{25} as a guard bit. In both cases, each digit q_i is computed iteratively by a classical non-restoring division step in base 2 [4]. The number of iterations is thus either 25 or 26 and the last iteration is followed by the usual correction step in case of a negative remainder. Note that in the case of 26 iterations, normalization of the result mantissa m_r implies that the result biased exponent $e_{r,b}$ is decreased by one. The remainder is then used to update the value of the sticky bit, and rounding of the mantissa can be done using both the guard and sticky bits.

In the case of denormals, we add the necessary pre-treatment and post-treatment operations. Again, special values are handled efficiently as described in phase B (Section 4).

Table 4 reports the timing, code size and bundle fill rate for the division operation.

	original lib.	new lib. wo/D	new lib. w/D
time (# bundle)	171 (100%)	117 (68%)	131 (77%)
code size	784 (100%)	792 (101%)	1232 (157%)
bundle fill rate	52%	65%	60%

Table 4: Main characteristics of the division operation.

4.4 Floating-Point Square Root

For $r = \sqrt{x}$ with r, x as in Section 2.1, one takes $s_r = s_x$ and $e_{r,b} = \lfloor (e_{x,b} + 127)/2 \rfloor$. If $e_{x,b}$ is even (that is, e_x is odd), we replace m_x with $2m_x$. Hence, for a normalized x, this new m_x satisfies $1 \leq m_x < 4$ and thus in any case $1 \leq \sqrt{m_x} < 2$. We then compute the first 24 fractional bits of $\sqrt{m_x} = 1.q_1q_2\cdots q_{23}q_{24}\cdots$ (with q_{24} as a guard bit) by a standard non-restoring iterative scheme [4]. As for division, these iterations are followed by a correction step based on whether the remainder is zero or not. This remainder further gives the value of the sticky bit which is updated before rounding the computed mantissa.

Notice that since square root is a unary operator, handling special values is simpler than for other binary operators; it can be implemented efficiently as follows. Let x be an operand encoded as the 32-bit unsigned integer `X` of Section 2.1. If `X` $> 2^{31}$ then a NaN is returned, because in this case x is nonzero, with negative sign. Otherwise, one detects whether $x \in \{\pm 0, +\infty, \mathrm{NaN}\}$ as before by testing if $e_{x,b} \in \{0, 255\}$; in this case, x is returned.

Table 5 reports the timing, code size and bundle fill rate for the square-root operation.

	original lib.	new lib. wo/D	new lib. w/D
time (# bundle)	123 (100%)	107 (87%)	113 (92%)
code size	448 (100%)	408 (91%)	496 (111%)
bundle fill rate	65%	82%	78%

Table 5: Main characteristics of the square root operation.

5 VALIDATION

Our library has been validated using two levels of test. The first level validates the individual operations. It is based of long vectors of chosen patterns for each operation. A test pattern is composed of the chosen argument(s) and the expected result of the operation. The operation *passes* the vector test if for each pattern the computed result is the expected value. The patterns are chosen to test all the branches in the algorithm as well as the behavior of the basic blocks of the algorithm. We developed some vectors for each operation, each rounding mode and for computations with and without the denormal numbers. The patterns include combinations of special values (NaN, $\pm\infty$, ± 0), some representable values and values that lead to difficult rounding cases. The vectors include exhaustive patterns for the special values. Some random values as well as representative values of the different bit fields $(\min, \max, \min+1, \max-1, (\max-\min)/2, \ldots)$ are used as representable values. Figure 4 presents an example of difficult rounding case for multiplication and rounding to nearest even (RNE) mode. The product $x \times y$ with $x = 76$ and $y = 883013$ (x and y are exactly representable in SP FP) should give a mathematical result of 67108988, but due to RNE mode on the SP format, the returned FP value must be 67108992.

The second level of validation of our library is based on the run of some numerical algorithms with known results. This second level validates the overall numerical behavior of the library. The following algorithms have been implemented and successfully tested on our FP library. The code name and the summary of each program is given below.

`DP`: Dot product $P = [1, 2, \ldots, n].[n, \ldots, 2, 1]^T$ for various n ($P = \sum_{i=1}^{n} i(n-i+1) = n^3/6 + n^2/2 + n/3$).

`REC`: Numerical integration using the rectangles method. For instance numerically evaluate $I = \int_1^2 x^{-1} dx$ (theoretically, $I = \ln 2 \approx 0.6931471$).

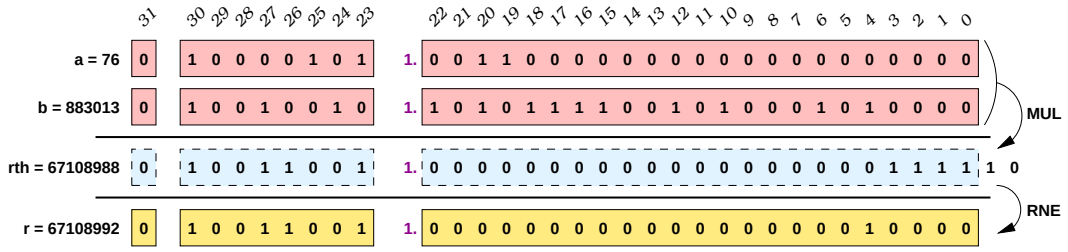

Figure 4: Validation example: round to nearest even of 76×883013.

`RK4`: Numerical solving of differential systems using Runge-Kutta order-4 method. For instance numerically solve $y' = x/4 - y/4 + 2$ with $y(0) = 0$ (the solution is $y(x) = x + 4 - 4e^{-x/4}$) and for various numbers of intermediate points.

`GAU`: Linear system solving using Gauss elimination (with partial pivoting). For instance, solve a $n \times n$ system for which the solution vector is $[1, 2, 3, \ldots, n]^T$.

6 PERFORMANCE MEASUREMENT

Some performance measurements have been done on the algorithms presented in Section 5 (`DP`, `REC`, `RK4` and `GAU`). The goal here is to study the impact of the different versions of the new library on complete typical applications. For each algorithm, we have used several values of the parameters (number of points, size of vectors and matrices). Here, we report average values of the complete measurements. In Figure 5 the relative computation time are reported. Those values are relative computation timings with respect to the original library performance.

In Figure 5, a $11-14\%$ speed improvement is achieved for the new library without denormal numbers (wo/D). For the new version with denormal numbers (w/D), the improvement is about $5-9\%$.

We also measure the code size of all the algorithms and for the different versions of the library. The impact of the new libraries on the complete code size is very small. In the case of the new library without denormal numbers, an average increase of the code size of about 0.3% is measured (with respect to the original library). The increase is about 0.8% (up to 1.4% in some cases) for the new version with denormal numbers. This shows that a significant increase of the code size for the individual operations has only a very small impact on the complete code size. Indeed, those functions are just inserted in only one place in the complete code. This should not lead to worse instruction cache performances.

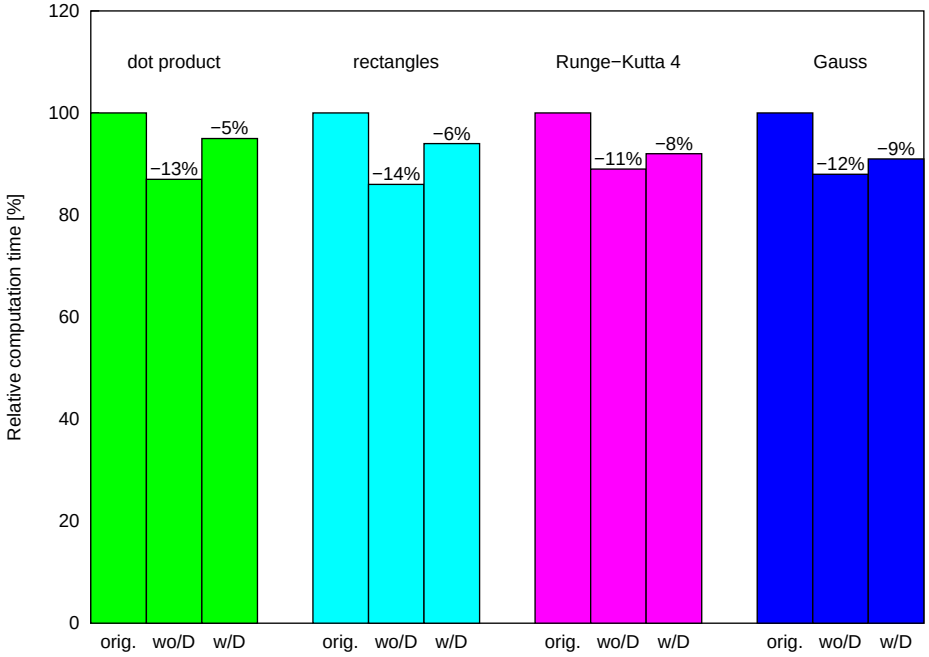

Figure 5: Comparison of relative computation timings for the various validation algorithms.

7 CONCLUSION & FUTURE PROSPECTS

In this paper, we have presented a new C library for single precision IEEE floating-point for processors without FP hardware units such as VLIW or DSP processor cores for embedded applications. This library provides the basic arithmetic operations (addition, subtraction, multiplication, division and square root) and quasi-fully complies with the IEEE 754 standard (the flags are not supported). In particular, this library offers the four rounding modes and supports denormal numbers. The library has been optimized and tested on STMicroelectronics processors of the ST200 family.

Also, the library has been validated at two different levels. Special values, specific numerical values and random values are used in the first level of validation. The cases involving special values such as ± 0, $\pm\infty$ or NaN have been tested exhaustively. Second, the library has been used successfully as the underlying arithmetic layer when running various numerical algorithms with known results.

The performances of the new library have been compared to those of the original library. The conclusions of the simulations done with this architecture are:

- When restricting to rounding to nearest and to normalized numbers, the individual operations of the new library are typically 30% faster than the original ones. The improvement on complete standard numerical applications is about $11 - 14\%$.
- Supporting the four rounding modes does not induce any significant slowdown.
- The code size of the individual operations is up to 50% larger. But for complete applications, this increase is limited to less than 1%. So, the instruction cache performances should not be changed.

Although our library already gives evidence that handling denormals can be done fairly efficiently, a first direction for future research will be to optimize it and speed it up even further. A second direction is to extend the set of available operations to other highly common algebraic operations such as x^2, $xy \pm z$, $1/x$, $1/\sqrt{x}$ and $1/\sqrt{x^2+y^2}$.

ACKNOWLEDGMENT

The financial support of the French *Région Rhône-Alpes* within the *"Arithmétique Flottante pour circuits DSP"* project is gratefully acknowledged.

References

[1] B. Dupont de Dinechin, C. Monat, P. Blouet, and C. Bertin. DSP-MCU processor optimization for portable applications. *Microelectronic Engineering*, 54(1–2):123–132, December 2000.

[2] Daniel Menard and Olivier Sentieys. Automatic evaluation of the accuracy of fixed-point algorithms. In *Design, Automation and Test in Europe (DATE)*, pages 529–537, 2002.

[3] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985*, 1985.

[4] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[5] P. Faraboschi, G. Brown, J. A. Fisher, G.Desoli, and F. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *27th Annual International Symposium on Computer Architecture – ISCA'00*, June 2000.

[6] HP and STMicroelectronics launch "LX". `http://www.embedded.com/2000/0010/0010feat6.htm`.

[7] Open64 compiler tools. `http://open64.sourceforge.net/`.

[8] B. Dupont de Dinechin, F. de Ferrière, C. Guillon, and A. Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems – CASES*, November 2000.

[9] Multiflow computer, VLIW, history, information. `http://www.reservoir.com/vliw.php`.

[10] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure placement using temporal-ordering information : dealing with code size expansion. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, September 2004.

[11] European telecommunications standards institute–ETSI. `http://www.etsi.org`.

[12] International telecommunication union–ITU. `http://www.itu.int`.

[13] DGL the data generation language. `http://cs.ecs.baylor.edu/~maurer/`.

[14] SoftFloat. `http://www.jhauser.us/arithmetic/SoftFloat.html`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399